Lesson 5

• Generation of Lexical Analyzers
Creating a Lexical Analyzer with Lex and Flex

lex source program lex.l

lex (or flex)

lex.yy.c

C compiler

a.out

input stream

a.out

sequence of tokens
Lex Specification

• A *lex specification* consists of three parts:
  *regular definitions, C declarations in % {  %} % *
  *translation rules % % *
  *user-defined auxiliary procedures *

• The *translation rules* are of the form:
  \[ p_1 \{ \text{action}_1 \} \]
  \[ p_2 \{ \text{action}_2 \} \]
  \[ \ldots \]
  \[ p_n \{ \text{action}_n \} \]
Regular Expressions in Lex

\* match the character \*  
\+ match the character .  
```
picture
  "string" match contents of string of characters
  . match any character except newline
  ^ match beginning of a line
  $ match the end of a line
  [xyz] match one character x, y, or z (use \ to escape -)
  [^xyz] match any character except x, y, and z
  [a-zA-Z] match one of a to z
  r* closure (match zero or more occurrences)
  r+ positive closure (match one or more occurrences)
  r? optional (match zero or one occurrence)
  r1r2 match r1 then r2 (concatenation)
  r1|r2 match r1 or r2 (union)
  ( r ) grouping
  r1\r2 match r1 when followed by r2
  \{ d \} match the regular expression defined by d
```
Example Lex Specification 1

Translation rules

```c
%{
#include <stdio.h>
%
%%
[0-9]+ { printf("%s\n", yytext); } 
. | \n { }
%%
main()
{ yylex(); }
}
```

Contains the matching lexeme

Invokes the lexical analyzer

```
lex spec.l
gcc lex.yy.c -ll
./a.out < spec.l
```
Example Lex Specification 2

Translation rules

```
 %{  
    #include <stdio.h>  
    int ch = 0, wd = 0, nl = 0;  
%}

delim   [ \t]+  
%
\n   { ch++; wd++; nl++; }
^{delim}   { ch+=yyleng; }
{delim}   { ch+=yyleng; wd++; }
.   { ch++; }
%
main()
{   yylex();
    printf("%8d%8d%8d\n", nl, wd, ch);
}
```

Regular definition
Example Lex Specification 3

```c
{%
#include <stdio.h>
%

digit     [0-9]
letter    [A-Za-z]
id        {letter}{letter}|{digit})*
%
{digit}+  { printf("number: %s\n", yytext); }
{id}      { printf("ident: %s\n", yytext); }
.         { printf("other: %s\n", yytext); }
%
main()
{ yylex(); }
```
Example Lex Specification 4

```c
{% /* definitions of manifest constants */
#define LT (256)
...
%
 delim [ \t\n]
 ws {delim}+
 letter [A-Za-z]
 digit [0-9]
 id {letter}({letter}|{digit})*
 number {digit}+(/\.{digit}+)?(E[+\-]?{digit}+)?
%
 {ws} { }
 if {return IF;}
 then {return THEN;}
 else {return ELSE;}
{id} {yyval = install_id(); return ID;}
{number} {yyval = install_num(); return NUMBER;}
<" {yyval = LT; return RELOP;}
<=" {yyval = LE; return RELOP;}
=" {yyval = EQ; return RELOP;}
<>" {yyval = NE; return RELOP;}
">" {yyval = GT; return RELOP;}
">=" {yyval = GE; return RELOP;}
%
int install_id()
...
%
```

Return token to parser

Token attribute

Install `yytext` as identifier in symbol table
Design of a Lexical Analyzer Generator

- Translate regular expressions to NFA
- Translate NFA to an efficient DFA

Diagram:

1. Regular expressions
2. NFA
3. DFA
   - Simulate NFA to recognize tokens
   - Simulate DFA to recognize tokens

Optional
Nondeterministic Finite Automata

• An NFA is a 5-tuple \((S, \Sigma, \delta, s_0, F)\) where

  \(S\) is a finite set of states
  \(\Sigma\) is a finite set of symbols, the alphabet
  \(\delta\) is a mapping from \(S \times \Sigma\) to a set of states
  \[
  \delta : S \times \Sigma \rightarrow P(S)
  \]
  \(s_0 \in S\) is the start state
  \(F \subseteq S\) is the set of accepting (or final) states
Transition Graph

• An NFA can be diagrammatically represented by a labeled directed graph called a transition graph

\[
S = \{0, 1, 2, 3\} \\
\Sigma = \{a, b\} \\
s_0 = 0 \\
F = \{3\}
\]
Transition Table

• The mapping \( \delta \) of an NFA can be represented in a *transition table*

\[
\begin{align*}
\delta(0, a) &= \{0, 1\} \\
\delta(0, b) &= \{0\} \\
\delta(1, b) &= \{2\} \\
\delta(2, b) &= \{3\}
\end{align*}
\]

<table>
<thead>
<tr>
<th>State</th>
<th>Input ( a )</th>
<th>Input ( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0, 1}</td>
<td>{0}</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>{2}</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>{3}</td>
</tr>
</tbody>
</table>
The Language Defined by an NFA

• An NFA *accepts* an input string $x$ (over $\Sigma$) if and only if there is some path with edges labeled with symbols from $x$ in sequence from the start state to some accepting state in the transition graph.

• A state transition from one state to another on the path is called a *move*.

• The *language defined by* an NFA is the set of input strings it accepts.

• What is the language accepted by the example NFA? 
  $\text{– (a}\mid\text{b})^{*}\text{abb}$
Design of a Lexical Analyzer Generator: RE to NFA to DFA

Lex specification with regular expressions

\[
p_1 \{ \text{action}_1 \} \\
p_2 \{ \text{action}_2 \} \\
\ldots \\
p_n \{ \text{action}_n \}
\]

NFA

\[
\begin{align*}
N(p_1) & \xrightarrow{\varepsilon} \text{action}_1 \\
N(p_2) & \xrightarrow{\varepsilon} \text{action}_2 \\
\ldots & \\
N(p_n) & \xrightarrow{\varepsilon} \text{action}_n
\end{align*}
\]

Subset construction

DFA
From Regular Expression to NFA (Thompson’s Construction)
An example:
RE -> Parse Tree -> NFA

(a | b)*abb
Combining the NFAs of a Set of Regular Expressions

\( a \{\ action_1 \} \)

\( abb \{\ action_2 \} \)

\( a*b+ \{\ action_3 \} \)
Simulating the Combined NFA
Example 1

![Diagram of NFA]

- **start**
- **0**
  - a → **1**
  - ε → **3**
  - ε → **7**
- **1**
  - a → **2**
  - ε → **1**
- **2**
  - a → **4**
  - b → **5**
  - b → **6**
- **3**
  - a → **4**
- **4**
  - b → **5**
- **5**
  - b → **6**
- **6**
- **7**
  - a → **8**
  - b → **1**
- **8**
  - b → **8**
- **9**
  - a → **8**

**Action**:
- **Action 1**: a → 2
- **Action 2**: a → 4, b → 5, b → 6
- **Action 3**: a → 8, none

**Instructions**:
- Must find the *longest match*:
- Continue until no further moves are possible
- When last state is accepting: execute action

1. **0** → a → **2**
2. **2** → a → **7**
3. **7** → b → **8**
4. **8** → a → none

**Action 3**: None
Simulating the Combined NFA
Example 2

When two or more accepting states are reached, the first action given in the Lex specification is executed.
Deterministic Finite Automata

• A deterministic finite automaton is a special case of an NFA
  – No state has an ε-transition
  – For each state $s$ and input symbol $a$ there is at most one edge labeled $a$ leaving $s$

• Each entry in the transition table is a single state
  – At most one path exists to accept a string
  – Simulation algorithm is simple
Example DFA

A DFA that accepts \((a \mid b)^*abb\)
Conversion of an NFA into a DFA

• The **subset construction algorithm** converts an NFA into a DFA using:
  \[
  \varepsilon\text{-}closure(s) = \{s\} \cup \{t \mid s \xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon} t\}
  \]
  \[
  \varepsilon\text{-}closure(T) = \bigcup_{s \in T} \varepsilon\text{-}closure(s)
  \]
  \[
  move(T, a) = \{t \mid s \xrightarrow{a} t \text{ and } s \in T\}
  \]

• The algorithm produces:
  * **Dstates** is the set of states of the new DFA consisting of sets of states of the NFA
  * **Dtran** is the transition table of the new DFA
**ε-closure and move Examples**

ε-closure({0}) = {0,1,3,7}
move({0,1,3,7},a) = {2,4,7}
ε-closure({2,4,7}) = {2,4,7}
move({2,4,7},a) = {7}
ε-closure({7}) = {7}
move({7},b) = {8}
ε-closure({8}) = {8}
move({8},a) = Ø

Also used to simulate NFAs (!)
Simulating an NFA using \( \varepsilon \)-closure and \textit{move} \\

\[
S := \varepsilon \text{-closure}(\{s_0\}) \\
S_{\text{prev}} := \emptyset \\
a := \text{nextchar}() \\
\textbf{while } S \neq \emptyset \textbf{ do} \\
\quad S_{\text{prev}} := S \\
\quad S := \varepsilon \text{-closure}(\text{move}(S,a)) \\
\quad a := \text{nextchar}() \\
\textbf{end do} \\
\textbf{if } S_{\text{prev}} \cap F \neq \emptyset \textbf{ then} \\
\quad \text{execute action in } S_{\text{prev}} \\
\quad \text{return “yes”} \\
\textbf{else} \\
\quad \text{return “no”}
The Subset Construction Algorithm: from a NFA to an equivalent DFA

- Initially, $\varepsilon$-closure($s_0$) is the only state in $Dstates$ and it is unmarked

while there is an unmarked state $T$ in $Dstates$ do
  mark $T$
  for each input symbol $a \in \Sigma$ do
    $U := \varepsilon$-closure(move($T, a$))
    if $U$ is not in $Dstates$ then
      add $U$ as an unmarked state to $Dstates$
    end if
    $Dtran[T, a] := U$
  end do
end do
Subset Construction Example 1

A = \{0,1,2,4,7\}
B = \{1,2,3,4,6,7,8\}
C = \{1,2,4,5,6,7\}
D = \{1,2,4,5,6,7,9\}
E = \{1,2,4,5,6,7,10\}
Subset Construction Example 2

\[
\begin{align*}
A &= \{0,1,3,7\} \\
B &= \{2,4,7\} \\
C &= \{8\} \\
D &= \{7\} \\
E &= \{5,8\} \\
F &= \{6,8\}
\end{align*}
\]

\textit{Dstates}
Minimizing the Number of States of a DFA
From Regular Expression to DFA Directly

• The “important states” of an NFA are those without an $\varepsilon$-transition, that is if $\text{move}([s], a) \neq \emptyset$ for some $a$ then $s$ is an important state

• The subset construction algorithm uses only the important states when it determines $\varepsilon$-closure($\text{move}(T, a)$)
What are the “important states” in the NFA built from Regular Expression?

- \( \varepsilon \)
- \( a \)
- \( r_1 \mid r_2 \)
- \( r_1 r_2 \)
- \( r^* \)
From Regular Expression to DFA Directly (Algorithm)

• The only accepting state (via the Thompson algorithm) is not important

• Augment the regular expression $r$ with a special end symbol # to make accepting states important: the new expression is $r#$

• Construct a syntax tree for $r#$

• Attach a unique integer to each node not labeled by $\varepsilon$
From Regular Expression to DFA
Directly: Syntax Tree of \((a|b)^{*}abb\#\)

concatenation
“cat-nodes”
closure
“star-node”
alternation
“or-node”
position
number
(for leafs ≠ε)
From Regular Expression to DFA Directly: Annotating the Tree

- Traverse the tree to construct functions \textit{nullable}, \textit{firstpos}, \textit{lastpos}, and \textit{followpos}
- For a node $n$, let $L(n)$ be the language generated by the subtree with root $n$
- $\text{nullable}(n): L(n)$ contains the empty string $\varepsilon$
- $\text{firstpos}(n):$ set of positions under $n$ that can match the first symbol of a string in $L(n)$
- $\text{lastpos}(n):$ the set of positions under $n$ that can match the last symbol of a string in $L(n)$
- $\text{followpos}(i):$ the set of positions that can follow position $i$ in the tree
From Regular Expression to DFA
Annotating the Syntax Tree of \((a | b)^* a b b \#$
From Regular Expression to DFA Directly: Annotating the Tree

<table>
<thead>
<tr>
<th>Node $n$</th>
<th>$\text{nullable}(n)$</th>
<th>$\text{firstpos}(n)$</th>
<th>$\text{lastpos}(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf $\varepsilon$</td>
<td>true</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Leaf $i$</td>
<td>false</td>
<td>${i}$</td>
<td>${i}$</td>
</tr>
<tr>
<td>$\mid$ \ $\mid$ \ $\mid$</td>
<td>$\text{nullable}(c_1)$ or $\text{nullable}(c_2)$</td>
<td>$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$</td>
<td>$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$</td>
</tr>
<tr>
<td>$\cdot$ \ $\cdot$ \ $\cdot$</td>
<td>$\text{nullable}(c_1)$ and $\text{nullable}(c_2)$</td>
<td>$\text{if } \text{nullable}(c_1) \text{ then } \text{firstpos}(c_1) \cup \text{firstpos}(c_2) \text{ else } \text{firstpos}(c_1)$</td>
<td>$\text{if } \text{nullable}(c_2) \text{ then } \text{lastpos}(c_1) \cup \text{lastpos}(c_2) \text{ else } \text{lastpos}(c_2)$</td>
</tr>
<tr>
<td>$\ast$</td>
<td>true</td>
<td>$\text{firstpos}(c_1)$</td>
<td>$\text{lastpos}(c_1)$</td>
</tr>
</tbody>
</table>
From Regular Expression to DFA
Directly: followpos

for each node $n$ in the tree do
  if $n$ is a cat-node with left child $c_1$ and right child $c_2$ then
    for each $i$ in lastpos($c_1$) do
      followpos($i$) := followpos($i$) \cup firstpos($c_2$)
    end do
  else if $n$ is a star-node
    for each $i$ in lastpos($n$) do
      followpos($i$) := followpos($i$) \cup firstpos($n$)
    end do
  end if
end do
From Regular Expression to DFA

followpos on the Syntax Tree of \((a\|b)^*abb\#\)

<table>
<thead>
<tr>
<th>NODE</th>
<th>(n)</th>
<th>followpos((n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>{4}</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>{5}</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>{6}</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>
From Regular Expression to DFA
Directly: Algorithm

\[ s_0 := \text{firstpos}(\text{root}) \text{ where } \text{root} \text{ is the root of the syntax tree for } (r)\# \]
\[ D\text{states} := \{s_0\} \text{ and is unmarked} \]

\textbf{while} there is an unmarked state } T \text{ in } D\text{states} \textbf{do}

\hspace{1em} \text{mark } T

\hspace{1em} \textbf{for} each input symbol } a \in \Sigma \text{ do}

\hspace{2em} \text{let } U \text{ be the union of } \text{followpos}(p) \text{ for all positions } p \text{ in } T
\hspace{2em} \text{such that the symbol at position } p \text{ is } a

\hspace{2em} \textbf{if } U \text{ is not empty and not in } D\text{states} \textbf{then}

\hspace{3em} \text{add } U \text{ as an unmarked state to } D\text{states}

\hspace{2em} \textbf{end if}

\hspace{2em} D\text{tran}[T, a] := U

\hspace{1em} \textbf{end do}

\textbf{end do}
From Regular Expression to DFA Directly: Example

<table>
<thead>
<tr>
<th>Node</th>
<th>followpos</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a {1, 2, 3}</td>
</tr>
<tr>
<td>2</td>
<td>b {1, 2, 3}</td>
</tr>
<tr>
<td>3</td>
<td>a {4}</td>
</tr>
<tr>
<td>4</td>
<td>b {5}</td>
</tr>
<tr>
<td>5</td>
<td>b {6}</td>
</tr>
<tr>
<td>6</td>
<td># -</td>
</tr>
</tbody>
</table>

Diagram of DFA with transitions:
- From state 1, on input 'b', go to state 3.
- From state 2, on input 'b', go to state 5.
- From state 3, on input 'b', go to state 4.
- From state 4, on input 'b', go to state 5.
- From state 5, on input 'b', go to state 6.
- From state 6, on input 'a', go back to state 5.